Rod Cutting

- Assume that a piece of rod of length i can be sold for price p_i for $1 \leq i \leq n$
- Define the *value* of a rod of length i as the maximum, over all possible ways to cut up the rod into integer-length pieces, of the sum of the selling prices of the pieces
- Determine an optimal way to cut up a given rod of length n

Brute Force Approach

- The number of different ways to cut up the rod corresponds to what number theorists call the number of "partitions" of n, denoted $p(n)$
- It is easy to see that $p(n) = \Omega(2^{\sqrt{n}})$
- It is known that $\ln p(n) \sim C\sqrt{n}$ where $C = \pi\sqrt{2/3}$

Towards an Efficient Algorithm

- Let v_i denote the value of a rod of length i, $0 \leq i \leq n$
- Suppose we have already determined $v_0, \ldots, v_{i-1}$ for some $i \geq 1$, and we now wish to determine v_i
- When cutting a rod of length i, if one of the pieces we create is required to be of length j, then the highest value we can achieve is $p_j + v_{i-j}$
- By maximizing the above expression over all j ranging from 1 to i, we can compute v_i in $O(i)$ operations

A Dynamic Programming Solution

- We have

$$v_i = \begin{cases} 0 & \text{if } i = 0 \\ \max\limits_{1 \le j \le i} p_j + v_{i-j} & \text{if } i > 0 \end{cases}$$

- We can compute v_0 to v_n in $O(n^2)$ operations
- We can "trace backwards" in the table of v_i values to compute an optimal solution in $O(n^2)$ operations
  - If we augment our table by also storing an "optimal j-value" for each i, we can recover an optimal solution in $O(n)$ operations

Matrix Chain Multiplication

- Suppose A is an $r \times s$ matrix and B is an $s \times t$ matrix
  - The cost of computing the matrix product $A \cdot B$ using elementary matrix multiplication is $\Theta(rst)$
  - For simplicity, let's consider this cost to be exactly rst
- We are given a positive integer n and a sequence of positive integers $p_0, \ldots, p_n$
- Determine a minimum-cost parenthesization of the matrix product $A_1 \cdot \ldots \cdot A_n$ where A_i is a $p_{i-1} \times p_i$ matrix for $1 \leq i \leq n$

Brute Force Approach

- It is easy to see that there are exponentially many parenthesizations of n factors
- The number of parenthesizations of n factors is equal to C_{n-1}, where C_k denotes the kth Catalan number
- It is known that

$$C_k = \frac{1}{k+1}\binom{2k}{k} \sim \frac{4^k}{\sqrt{\pi}k^{3/2}}$$

Towards an Efficient Algorithm

- For $1 \leq i < j \leq n$, let $a_{i,j}$ denote the cost of an optimal parenthesization of $A_i \cdot \ldots \cdot A_j$
- In any parenthesization of $A_i \cdot \ldots \cdot A_j$ where $1 \leq i < j \leq n$, there is an integer k such that $i \leq k < j$ and the last operation performed computes the product of the $p_{i-1} \times p_k$ matrix $A_i \cdot \ldots \cdot A_k$ with the $p_k \times p_j$ matrix $A_{k+1} \cdot \ldots \cdot A_j$
  - For a given choice of k, the lowest cost we can achieve is thus $a_{i,k} + a_{k+1,j} + p_{i-1}p_k p_j$
  - By minimizing the preceding expression over all possible choices of k, we can compute $a_{i,j}$
  - Note that $k - i$ and $j - (k + 1)$ are each less than $j - i$ since $i \leq k < j$

A Dynamic Programming Algorithm

- We have

$$a_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} a_{i,k} + a_{k+1,j} + p_{i-1} p_k p_j & \text{if } i < j \end{cases}$$

- We can compute the table of $a_{i,j}$ values in $O(n^3)$ operations
- We can trace backwards in the table of $a_{i,j}$ values to determine an optimal solution in $O(n^2)$ operations
  - If we augment our table by also storing an "optimal k-value" for each pair (i, j), we can recover an optimal solution in $O(n)$ operations

Longest Common Subsequence

- A subsequence of a sequence Z is any sequence obtainable from Z by removing zero or more elements of Z
- We are given two sequences $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$
- A longest common subsequence (LCS) of X and Y is a longest sequence that is a subsequence of both X and Y
- It is easy to see that the brute force approach requires an exponential number of operations

Towards an Efficient Algorithm

- For all i and j such that $0 \leq i \leq m$ and $0 \leq j \leq n$, let $a_{i,j}$ denote the length of an LCS of $X_i = x_1, \ldots, x_i$ and $Y_j = y_1, \ldots, y_j$
- If $i = 0$ or $j = 0$, it is easy to compute an LCS of X_i and Y_j, so let's assume $i > 0$ and $j > 0$
- If $x_i = y_j$, then there is an LCS of X_i and Y_j that consists of an LCS of X_{i-1} and Y_{j-1} followed by the symbol $x_i = y_j$
- If $x_i \neq y_j$, then some LCS of X_i and Y_j is either an LCS of X_{i-1} and Y_j or an LCS of X_i and Y_{j-1}

A Dynamic Programming Algorithm

- We have

$$a_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ a_{i-1,j-1} + 1 & \text{if } i > 0 \text{ and } j > 0 \text{ and } x_i = y_j \\ \max(a_{i-1,j}, a_{i,j-1}) & \text{otherwise} \end{cases}$$

- We can compute all of the $a_{i,j}$ values in $O(mn)$ operations
- We can trace backwards in the table of $a_{i,j}$ values to obtain an optimal solution in $O(m+n)$ operations

The Knapsack Problem

- We are given a knapsack with positive integer weight capacity W, and n items indexed from 1 to n
  - Item i has positive integer weight w_i and positive integer value v_i
- We wish to find a maximum-value subset of the items that fit into the knapsack

A Dynamic Programming Algorithm

- For any integers i and j such that $0 \leq i \leq W$ and $0 \leq j \leq n$, let $a_{i,j}$ denote the value of a max-value subset of items 1 through j with weight at most i
- We have

$$a_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ a_{i,j-1} & \text{if } i > 0 \text{ and } j > 0 \text{ and } i < w_j \\ \max(a_{i,j-1}, v_j + a_{i-w_j,j-1}) & \text{otherwise} \end{cases}$$

- We can compute the $a_{i,j}$'s in $O(nW)$ operations
- We can trace backwards in the table of $a_{i,j}$ values to obtain an optimal solution in $O(n + W)$ operations

Comments on the $O(nW)$ Bound

- There are two parameters in this bound, n and W
  - There exist positive constants c, n_0, and W_0 such that for all $n \geq n_0$ and $W \geq W_0$, the number of operations is at most cnW
- Note that W could be large (e.g., $W = 2^n$), so this bound is not polynomial in the parameter n alone
  - It is polynomial in $n + W$
- Such multi-parameter bounds are commonplace
- Does the $O(nW)$ bound imply that our algorithm runs in "polynomial time"?

Polynomial Time Algorithms

- When we say that an algorithm runs in polynomial time, we mean that the running time is upper bounded by a polynomial in the length of the input (in bits, say)
  - Here we assume an efficient encoding of the input
  - For example, for a problem where the input is an integer k, an efficient encoding should use $\Theta(\log k)$ bits to represent k rather than a $\Theta(k)$-bit unary encoding
- For knapsack, we can consider the input length to be the sum of the lengths of the binary representations of each of the numbers in the input

Further Comments on the $O(nW)$ Bound

- Consider the family of all n-item knapsack instances where the length of the binary representation of each number in the input is $\Theta(n)$ bits
- Thus the length of the input is $\Theta(n^2)$ bits
- Since $W = 2^{\Theta(n)}$, the $O(nW)$ bound is not polynomial in the input size
- Hence the foregoing dynamic programming algorithm for knapsack is not a polynomial-time algorithm
- If we consider the variant of the knapsack problem in which the items weights are restricted to be at most polynomial in n (i.e., $O(\log n)$-bit numbers), then we have a polynomial-time algorithm

Pseudopolynomial Time

- An algorithm is said to run in pseudopolynomial time if its running time is polynomial in the length of the input when all of the integers in the input are represented in unary
- When all of the integers in the input are represented in unary, the length of a knapsack instance is $\Omega(n + W)$, and hence our $O(nW)$ bound is at most quadratic in the input size
- Thus our $O(nW)$ bound yields a pseudopolynomial time algorithm for knapsack
- Later in the course, we will see that the knapsack problem is an example of an NP-complete problem
  - Accordingly, no polynomial time algorithm for knapsack exists unless P equals NP (generally considered unlikely)

Another Dynamic Programming Algorithm for Knapsack

- Let V denote $\sum_{1 \leq j \leq n} v_j$
- For any integers i and j such that $0 \leq i \leq V$ and $0 \leq j \leq n$, let $b_{i,j}$ denote the weight of a min-weight subset of items 1 through j with value at least i (or ∞ if no such subset exists)
- We have

$$b_{i,j} = \begin{cases} 0 & \text{if } i = 0 \\ \infty & \text{if } i > 0 \text{ and } j = 0 \\ \min(b_{i,j-1}, w_j) & \text{if } i > 0 \text{ and } j > 0 \text{ and } i < v_j \\ \min(b_{i,j-1}, w_j + b_{i-v_j,j-1}) & \text{otherwise} \end{cases}$$

- We can compute the $b_{i,j}$'s in $O(nV)$ operations
- We can trace backwards in the table of $b_{i,j}$ values to obtain an optimal solution in $O(n + V)$ operations

Comments on the $O(nV)$ Bound

- Provides another pseudopolynomial time algorithm for knapsack
- If we consider the variant of the knapsack problem in which the item values are restricted to be at most polynomial in n (i.e., $O(\log n)$-bit numbers), then we have a polynomial-time algorithm

The Complexity of the Knapsack Problem: Summary

- If the item weights are $O(\log n)$-bit numbers, the knapsack problem can be solved in polynomial time
- If the item values are $O(\log n)$-bit numbers, the knapsack problem can be solved in polynomial time
- The (unrestricted) knapsack problem is solvable in pseudopolynomial time, and is not solvable in polynomial time unless P equals NP

A Space-Saving Technique for Certain Dynamic Programs

- A divide-and-conquer technique can be used to reduce the space requirement of many dynamic programming algorithms
- We will illustrate the technique in the context of the $O(nW)$ knapsack algorithm
- As previously described, this algorithm uses $O(nW)$ space
  - Remark: For the purposes of this discussion, we consider each input value and each table entry to occupy a single unit of space
- We will show how to reduce the space requirement to $O(n + W)$ while preserving the $O(nW)$ bound on the number of operations

Computing the Value of an Optimal Solution

- If we are only asked to compute the value of an optimal solution, it is easy to reduce the space requirement to $O(n + W)$
  - Our recurrence for the $a_{i,j}$'s allows us to compute column j of the table in terms of column $j - 1$ only
  - We only need space for the current column and the previous column
- But how can we obtain an optimal solution without keeping the entire table?

Any Optimal Solution Corresponds to a "Staircase"

- When we trace backwards in the full 2D table from entry $a_{W,n}$ to obtain an optimal solution, we visit exactly one entry in each column
- For $0 \leq j < n$, the entry that we visit in column j is an a row that is less than or equal to the entry that we visit in column $j+1$
- In this sense, the visited entries form a "staircase" pattern that goes from the bottom-left table entry $(0, 0)$ to the top-right table entry (W, n)
  - Remark: If entry (i, j) is on the staircase, then the prefix of the staircase terminating at (i, j) is an optimal staircase for the (i, j) subinstance
- To recover an optimal solution, it is enough to recover the corresponding staircase

Recovering the Middle Stair

- Assume for simplicity that $n = 2^k$ for some $k \geq 1$
- Suppose we only want to recover the "middle stair" of some optimal staircase
  - That is, we want to determine a row i such that entry $(i, n/2)$ of the table is the middle stair of some optimal staircase
- We can keep only the current and previous columns as before, but after we pass the middle column, we maintain two values in each table entry (i, j):
  - The value $a_{i,j}$, as before
  - The index of a row i' such that an optimal staircase for the knapsack subinstance corresponding to entry (i, j) includes table entry $(i', n/2)$

A Divide-and-Conquer Strategy

- So far, we have used $O(nW)$ time and $O(n + W)$ space to recover only the middle stair $(i^*, n/2)$ of an optimal staircase
  - We still need to recover the left and right halves of the staircase
  - The left half corresponds to the optimal staircase for the knapsack subinstance with capacity i^* and items 1 through $n/2$
  - The right half corresponds to the optimal staircase for the knapsack subinstance with capacity $W - i^*$ and items $(n/2) + 1$ through n
- Thus we can use recursion to recover the left and right halves

Space Usage

- As we determine entries in the optimal staircase, we store them in a global array of length n
- Each recursive call uses $O(W)$ scratch space to store the $O(1)$ columns needed to compute a middle stair
- This scratch space can be reused within each of the two recursive calls
- The total space usage remains $O(n + W)$

Analyzing the Number of Operations

- Let $T(W, k)$ denote the worst-case number of operations for any knapsack instance with capacity W and 2^k items
- There is a positive constant c such that $T(W, k) \leq c$ if $W = 0$ or $k = 0$, and otherwise $T(W, k)$ is at most

$$cW2^k + \max_{0 \leq i \leq W} [T(i, k-1) + T(W-i, k-1)]$$

- It is straightforward to prove by induction on $k \geq 0$ that

$$T(W, k) \leq c(3W2^k + 1)$$

holds for all $W \geq 0$

The Bellman-Ford Algorithm

- We are given a digraph $G = (V, E)$ where each edge (u, v) in E has an associated weight $w(u, v)$
- Edge weights may be negative, but there are no negative-weight cycles
- There is a designated source vertex s
- We wish to determine a shortest path from s to every vertex that is reachable from s
  - This is called the single source shortest paths (SSSP) problem
  - We'll focus on computing the shortest path distances; it is easy to modify the algorithm to obtain the shortest paths

Towards an Efficient Algorithm

- The length of a shortest path (i.e., the number of edges in the path) is at most $|V| - 1$
- For any vertex v in V and any integer k such that $1 \leq k < |V|$, we define $P(v, k)$ as the set of all minimum-weight paths of length at most k from s to v, and $d(v, k)$ as the weight of any path in $P(v, k)$ (or ∞ if $P(v, k)$ is empty)
- Suppose P is a positive-length path in $P(v, k)$
- If P has length less than k, then P belongs to $P(v, k-1)$ and $d(v, k) = d(v, k-1)$
- If P has length k, then the length-$(k-1)$ prefix of P belongs to $P(u, k-1)$ where u denotes the predecessor of v on P, and hence $d(v, k) = d(u, k-1) + w(u, v)$

A Dynamic Programming Algorithm

- Observe that $d(s, 0) = 0$ and $d(v, 0) = \infty$ for $v \neq s$
- For $1 \leq k < |V|$, we have

$$d(v, k) = \min(d(v, k-1), \min_{u \in V : (u,v) \in E} d(u, k-1) + w(u, v))$$

- It takes $O(|E|)$ operations to compute the $d(v, k)$'s from the $d(v, k-1)$'s
- We can compute all of the shortest path distances in $O(|E| \cdot |V|)$ operations

A Variation

- We can maintain a single "label" d'_v for each vertex v in V
- We initialize d'_s to 0 and d'_v to ∞ for $v \neq s$
- For k running from 1 to $|V| - 1$, we update each label d'_v to

$$\min(d'_v, \min_{u \in V:(u,v) \in E} d'_u + w(u, v))$$

- It is easy to prove (by induction on k) that after k iterations, we have $d'_v \geq d(v, |V| - 1)$ and $d'_v \leq d(v, k)$
- Thus, upon termination, the labels are equal to the shortest path distances